Chapter 4

Provenance Framework for Graph Databases

In the present digital era, everyday a huge amount of digital content is generated by various social media platforms mainly through mobile devices and shared with other users [89, 138, 156]. In this way, social media has been playing a major role in information sharing at a large scale due to easy access, low cost, and faster dissemination of information. Therefore, establishing the credibility of digital content spread across the social media platforms has attracted significant interest in research community. Social media's almost unlimited capacity to disseminate the information across ample audience has raised certain challenges such as fact investigation [162, 171, 166], network analytics [129], rumor centrality analysis, fake news detection [147], trustworthiness, data credibility [172] etc. Although, social media has a positive impact in expanding society's access to information as well as it quickly amplifies the influence of digital content. However, no one cares about the truthiness of that content. We no longer know where this content comes from & whether it is correct or not.? In this context, identification of the source of a digital content and its history has become a crucial challenge in the field of social media

- Asma Rani, Navneet Goyal, and Shashi K. Gadia. 2021. Provenance Framework for Twitter Data using Zero-Information Loss Graph Database. In 8th ACM IKDD CODS and 26th COMAD (CODS COMAD 2021). Association for Computing Machinery, New York, NY, USA, 74–82. DOI:https://doi.org/10.1145/3430984.3431014.

analytics. Provenance information can play a major role in addressing the challenges [78, 96, 2, 149, 9, 83, 30, 164].

In social media analytics, the accountability of an analysis largely relies on the data quality and trustworthiness of input data. ***Social provenance*** is often referred as the information about who created or changed content, what and how it was changed, regardless of their geographic location or access technology [156]. Social provenance of a published data is important for determining the amount of trust and the quality of data along with expected level of imprecision. It also plays a vital role in assessing authenticity of a piece of information published on a social media platform.

SQL & NoSQL Database: Relational databases have been the mainstay of the data community for decades starting from mid 1980's. They are ideal for structured data and predictable workload. Their schema is largely rigid, complex and require expensive vertical scaling from time to time. But, these databases are not scalable for handling Big Data which encompasses not just structured data, but also semi-structured and unstructured data. Not only this, the query workload is also unpredictable. Not Only SQL (NoSQL) databases have been proposed as an alternative to SQL databases to handle the challenges posed by big data, as these databases efficiently supports to a low latency, horizontal scalability, efficient storage, high availability, high concurrency, and reduced operational costs. SQL databases are purely based on set theory in which relational algebra and calculus are used for theoretical system design. These databases mainly focus on various set operations such as unions, intersections, difference, subsets etc., for transactions. While NoSQL databases rely on graph theory and focus on graph algebra for data analysis and decision support based on following mathematical properties viz. inverses, associativity, identities, commutativity, distributivity etc. These databases are designed to analyse sparse relations among data.

Evolution of NoSQL Databases: The scalability issues with relational databases due to exponentially growing data, were first identified by some big companies such as Facebook, Google, Amazon etc. They tried to fix their issues with their own solutions viz. BigTable, DynamoDB etc. This common interest resulted in several NoSQL database design with following feature i.e. consistency, performance, reliability etc. NoSQL represents a family of databases in which each database is quite different from others having

literally nothing in common. The only commonality is that they use a data model with structure that is different from the traditional row-column relation model of RDBMSs. Graph, Document, Column-oriented, & Key-value pair are the four kinds of NoSQL databases. The growth of unstructured data has led to interest in NoSQL databases. Nearly, 80% of the expected growth in data is in unstructured data. Social media platforms are the major source of unstructured data in the current times. Unstructured data is characterized by adhoc schema and therefore cannot be stored in SQL databases. NoSQL databases are much better suited due to their flexible schema. Text and multimedia are the most common forms of unstructured data. Although, there are over 150 different databases that belong to the NoSQL family, yet an increasing attention is being paid to the Graph and Key-Value Pair (KVP) database, as a class of non-relational database, and Neo4j and Apache Cassandra are most popular among them.

Therefore, in the thesis, we attempt to design and develop data provenance frameworks for SQL, Graph, and Key-value pair databases which collectively store approximately 90% of world's data. Provenance framework for relational database is already presented in Chapter 3. Provenance frameworks for graph and KVP databases are to be discussed in this chapter and next chapter respectively.

4.1 Introduction to Graph Database

Graph Database is a comparatively recent technology as compared to the established database technology like the relational database. Graph database uses a non-relational foundation. These are very flexible by nature (i.e., dynamic schema), extremely scalable, and focuses on relationships among relevant data. Graph databases are extremely fast for querying as compared to relational databases. In a graph database, real-world entities are represented as a node. A node can have attributes in the form of a key-value pair, and can be associated with other nodes via edges/relationships. Bidirectional relationship between two nodes can be modeled as two separate directed relationships, one in each direction. Nodes and edges with analogues attributes can be grouped under one label respectively. In contrary to relational database, a graph database is schema free and flexible, as every node in a graph may have different number of attributes and may be

associated with different number of nodes via explicit relationships. Graph databases natively embrace relationships thus suitable to store, process, and query graph-oriented data in efficient manner independent of size of a database, whereas relational databases need to compute relationships during query execution through complex join operations.

The key purpose of graph database is to efficiently store and query graph-oriented data such as social network data, network communications, neural networks, linked network, integrated circuits, road network etc. These databases are prominently used in research and analysis because they organize the relevant data using their relationships and quickly respond to complex queries. In current scenario, social networking sites such as Twitter, Facebook, Instagram, WhatsApp etc. has been playing a major role in disseminating information at global scale. In social media networks, each user can post his/her opinion, ideas, personal information, and other linked users such as friends/followers may like/dislike or comment on that post and may forward it also. This social media information is very complex in nature but can easily be represented and analyzed using graph databases.

In recent times, Twitter's popularity has sky rocketed as an enormous source of information, generating around 6000 tweets every second. It has led to research in various domains like fact investigation, rumor detection etc [162, 147, 171, 172, 129]. Researchers can get access to Twitter data through public APIs. On twitter, each user's profile is a rich source of information which contains User Id, User's Screen Name, User's real name, User's location, URL, Textual Descriptions, Number of Tweets Published, Profile Creation Date etc [99]. A user's tweet is a status message or a post displayed on user's profile with length up to 140 characters long, including Hashtags, user mentions, and URLs. A registered twitter user either can post its own tweet or can retweet the information that has been already posted by other users. A very large volume of tweets are generated by several twitter users for sharing some personal information or their opinion related to any specific event by tagging. The influence and popularity of a social media user can be measured by its audience in terms of friends or followers count.

Twitter's social network graph [155, 145, 137], is generally represented by a directed graph G(V,E) with multivariate properties, where V is a set of labeled nodes known as '*Entity*', E is a set of directed labeled edges known as '*Relationship*', and both V and

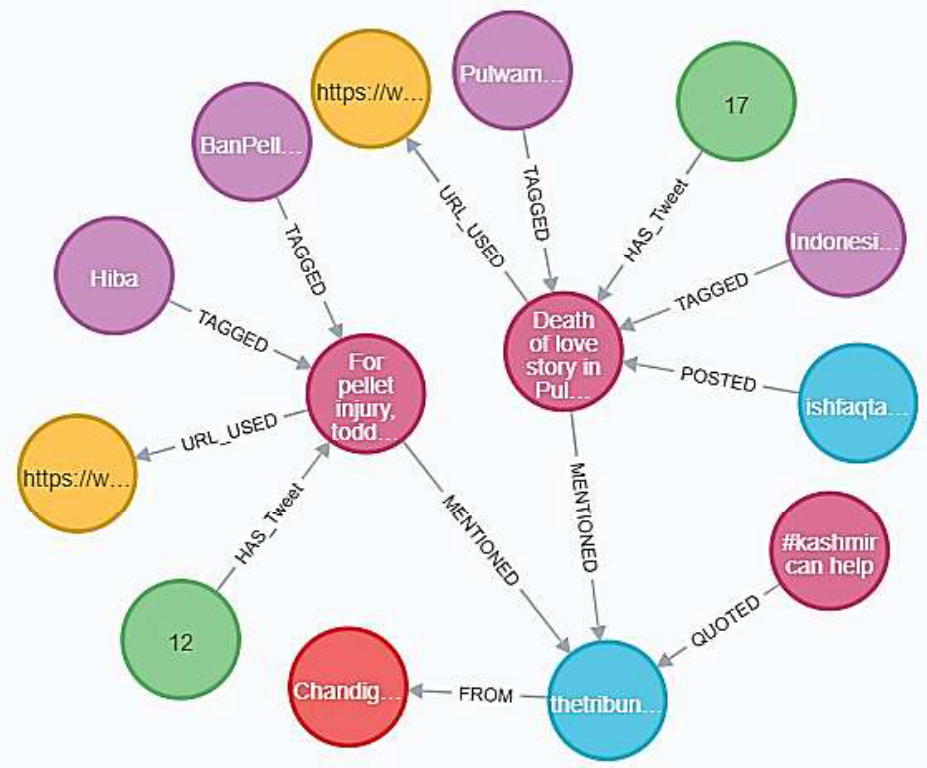


Figure 4.1: Twitter Social Network Graph

E can also have their own properties in the form of key-value pair. This multivariate nature of both nodes and edges makes a social network very complicated. The source of information is represented by a node 'v' and flow of information from one node to another is represented by an edge 'e', where v ∈ V and e ∈ E. In other words, we can say that V is the set of twitter users say v_1, v_2, v_3,...,v_n those produces the information and E is the set of relationships among them. A snapshot of dissemination of information generated by the Twitter is shown in Figure 4.1, where nodes are labeled with USER (Blue Nodes), COUNTRY (Red Nodes), TWEET (Pink Nodes), HASHTAG (Purple Nodes), URL (Yellow Nodes) etc. In a similar fashion, the directed edges represent the flow of information and are labeled with TAGGED, MENTIONED, REPLY_TO, POSTED, FROM, QUOTED etc.

4.2 Provenance in Graph Databases

Today's Social Networking Sites (SNS) such as Twitter, Facebook, Instagram etc. are based on the tenets of mobilization and de-contextualization of information, where each user stands at the edge of a river of information to pick an independent data object for

either repost or sharing with other users, without including ownership and description of the content. In such scenario, it is very crucial to perceive the origin and deriving process of a piece of information, and when such information was updated since its existence? The concept of social provenance [117, 152] can be used to answer the above questions. Social provenance is not only associated with a social media data's history in time, but also with the relationships of this data product with other entities which enabled its creation. It involves the following three dimensions; viz. "*What*", "*Who*", and "*When*". *What* provides descriptions about the social media posts, *Who* describes the correlations among social media users, and *When* characterizes the evolution of users' behavior over time. Now, it is clear from the above definitions that the whole idea of social provenance is about quality, credibility, and trust of a data product published on a social media platform. An enormous volume of data is produced rapidly and spread among a large number of people seamlessly across all kinds of boundaries. But, zero or very minimal provenance information is provided by such social networking sites to the users, which is not sufficient for auditing, investigations, knowing truthiness of contents etc. Consequently, a provenance capturing framework for social media environment is required to restore trust, and to improve information literacy [156].

From the literature reported in Chapter 1 on provenance frameworks in graph databases, it is obvious that most of the existing approaches for provenance in graph databases are not scalable to track provenance metadata for social media efficiently. Existing frameworks primarily focused on workflow provenance which captures coarse-grained information rather than detailed fine-grained provenance. Although, a few frameworks capture fine-grained provenance, but they do not support all type of queries. To the best of our knowledge, none of the existing frameworks support provenance for historical queries and provenance for data updates.

To bridge the above identified gaps, we propose a *Social Data Provenance (SDP)* framework based on the concept of Zero-Information Loss Database [3]. Qualitative analysis of existing provenance solutions and our proposed SDP framework based on an evaluation matrix is presented in Table 1.3 of Chapter 1 (page no. 35). Evaluation matrix includes provenance granularity level, type of queries supported for provenance generation, provenance visualization, and applicability. One of the main contributions of this work is to

design a *Zero-Information Loss Graph Database (ZILGDB)* which stores social media data along with time of its existence, time of expiry, and maintains information about all data updates. The proposed provenance framework is capable of capturing provenance for all queries including *select queries*, *aggregate queries*, *historical queries*, i.e., queries which were executed in the past. It is also suitable for capturing provenance information of all *update*, *delete*, and *insert* queries. The framework supports querying provenance information for tracking updates i.e. *historical data queries*, *justifying result of a query*, and *source of a query result*. The framework also supports efficient *multi-depth querying of provenance* data to explore direct/indirect sources of a piece of information. As a use case, we use the framework for investigating terrorist attack by identifying suspicious persons and their linked communities using social media data. The salient features of SDP framework are:

- ***Zero-Information Loss Graph Database (ZILGDB)***: ZILGDB is a special kind of temporal database which supports data versioning to maintain history of all the updates as provenance information along with the provenance of insertion and deletion operations.

- ***Provenance for Current Queries***: Proposed provenance framework enables querying the current snapshot of a data and captures the provenance for all the results of a query. The captured provenance information is stored in a Provenance Graph Database (PGDB) for querying and visualization.

- ***Provenance for Historical Queries***: Proposed framework supports past queries i.e. it traces the provenance of all the result tuples of a query executed in the past, for example consider the following query "Display all hashtags used by a specific user" executed on 19/10/2019.

- ***Querying Historical Data***: Proposed framework also allows querying historical data [136, 172] by introducing four new query constructs viz. *"instance"*, *"all"*, *"valid_on now"*, and *"valid_on 'date'"* as extended Cypher query constructs. It supports querying a data object with a given time in the past and with a time range specified in the query statement [148].

- ***Provenance Visualization***: Stored provenance information can be further analyzed

for various purposes such as justifying the result tuple of a query, querying historical data, audit-trail etc. via multi-depth provenance query. In section 4.6 of this chapter, provenance visualization has been applied to terrorist attack investigations.

4.3 ZILD Architecture for Graph Database (ZILGDB)

The proposed Social Data Provenance (SDP) framework is developed on top of Zero-Information Loss Graph Database (ZILGDB). ZILGDB is designed using the concept of Zero-Information Loss Database [3]. The key purpose to design ZILGDB is to maintain record of all update, delete, and insert operations without loosing any information, as provenance data. This information can be used further in querying provenance for various historical data queries,and for capturing provenance for historical queries. We state the problem of ZILGDB design as below:

Problem 1: *Zero-Information Loss Graph Database (ZILGDB) Generation Problem.*

ZILGDB generation problem is divided into three sub-problems based on the type of database operation i.e. insert query, update query or delete query.

1. Given a Graph G(V, E) and an Update Query Q_U to be executed on G. Our objectives are:

   (a) To find v, a_v, $G_v(v_a,e_a)$ where $v \in V$ is the vertex to be updated, a_v is the attribute of v to be updated, and $G_v(v_a,e_a)$ is a subgraph of G such that $v_a \in V$, $e_a \in E$ where v_a is set of vertices associated with v via edges e_a in G.

   (b) To create a new node as a clone of vertex v in Graph G, and associating it with all v_a via edges e_a as the original vertex v, but with the updated value of attribute a_v. Afterwards, setting "*valid_from*" attribute of new node as "*current date/time*", and "*valid_to*" attribute of old node v as "*current date/time*".

2. Given a Graph G(V,E) and Insert Query Q_I to be executed on G, our objective is to insert a new node and set the time of its existence i.e. "*valid_from*" attribute as "*current date/time*".

3. Given a Graph G(V,E) and Delete Query Q_D to be executed on G, our objectives are:

(a) To retrieve the node v to be deleted as per query Q_D.

(b) To update the node v with its time of expiry i.e. "*valid_to*" attribute as "*current date/time*".

The proposed ZILGDB architecture is shown in Figure 4.2. The architecture consists of following components viz. Query Parser, Query Rewriter, Query Generator, and Graph Database. When a user issues a query, the query is initially sent to the Query Parser (QP), where it is parsed and also type of the query is marked as Insert (I), Update (U), or Delete (D) query. If the issued query type is an "*Insert Query*" then the parsed results are sent to the Query Rewriter (QR) as mentioned in step I_1 and corresponding rewritten query (Q_I) is generated in step I_2. Here, the "*valid_from*" attribute of this new node is being set as a "*current date/time*" and then it is sent to the graph database for further execution. Now, if the issued query type is a "*Delete Query*" then the parsed results are sent to the Query Generator (QG) and the corresponding Update Query Q_U is generated to set "*current date/time*" as the value of "*valid_to*" attribute of the node to be deleted as shown in step D_1 and D_2 respectively, and then it is sent to graph database for execution. Similarly, if the issued query type is an "*Update Query*" then the parsed results are sent to both QG and QR in step U_{1a} and U_{1b} respectively. Then, in step U_2, corresponding Select Query (Q_S) generated from QG is executed on graph database, to retrieve an original node 'v' to be updated. After retrieving the node to be updated, in step U_3, corresponding Create

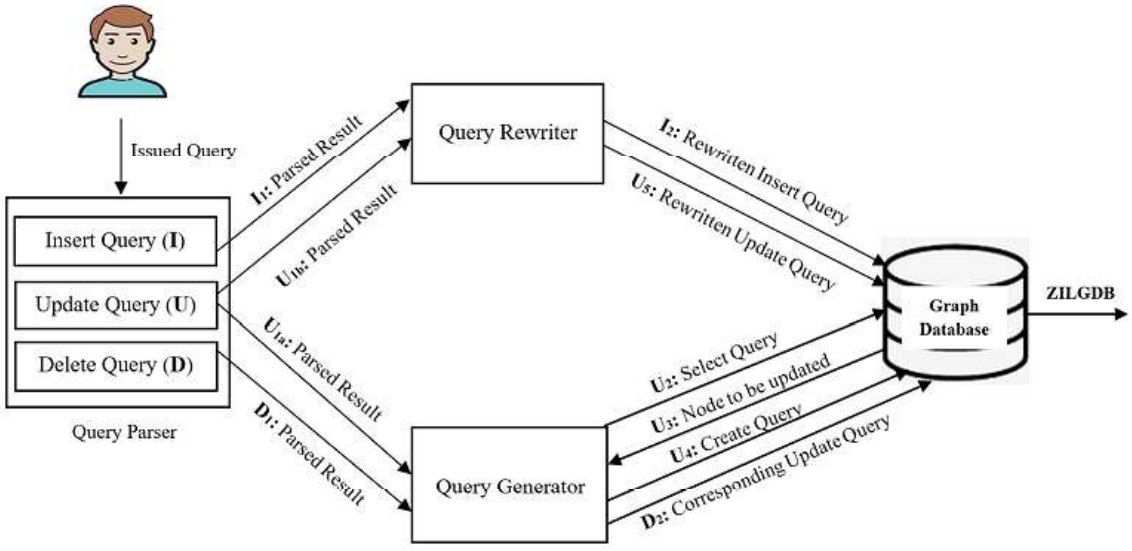


Figure 4.2: ZILGDB Architecture

Query (Q_C) is further generated by QG to create a new node 'v_c' that is a clone of 'v' and associating it with all nodes as 'v' via same type of edges, but with updated value of attribute as specified in original query. In the end, the Q_C and Rewritten Update Query (Q_{RU}) from QR, in steps U_4 and U_5 respectively, are executed on graph database that maintains history of update operation.

Pseudo-code for designing ZILGDB is mentioned in Algorithm 3. A "*valid_from*" attribute is added to every node whenever it is inserted (refer to lines 11 to 13). This attribute stores node's created time i.e. "*current date/time*" as a value. Whenever a delete operation is performed, a "*valid_to*" attribute is then added to the corresponding node with "*current date/time*" as value (refer to lines 15 and 16). It shows that the node has already expired but still exists in the database, which help in past queries execution to obtain the same result as their previous executions before delete operation occurs. Whenever an update operation is performed, initially the original node which is to be updated is retrieved

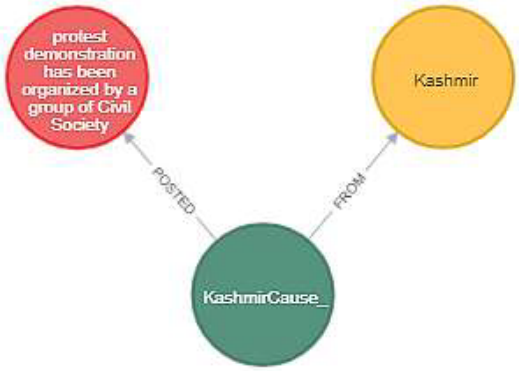
Algorithm 3 ***ZILGDB Construction***: Construct ZILGDB (Zero Information Loss Graph Database)

Input: Graph G(V, E), Query Q (Insert, Update, or Delete Query)

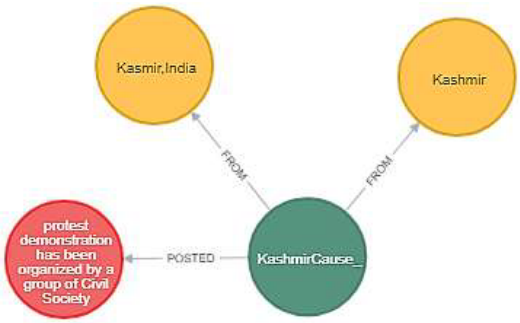
Output: Graph G(V, E) with history maintained

1: Parsed Result (P), Query Type (Q_T) ← Query Parser (Q)
2: **if** Q_T is **Update-Query then**
3: **Obtain** A_i and v_n ← P *// where A_i is the attribute to be updated, v_n is new //value of attribute A_i*
4: **Get** Select Query(Q_S) corresponding to Q, which will return the node needs to be updated
5: Vertex V_o ← Execute (G, Q_S) *// where $V_o \in V$, is a node to be updated*
6: Vertex V_u ← Clone V_o
7: Add Edges to V_u ← Clone Edges of V_o
8: Updated V_u ← Update A_i with v_n of V_u
9: Add Attribute "*valid_to*" to V_o with value as "*current date/time*"
10: Add Attribute "*valid_from*" to V_u with value as "*current date/time*"
11: **else if** Q_T is **Insert-Query then**
12: Q_I = **Get** Rewritten Insert Query by adding "*valid_from*" attribute with value as "*current date/time*"
13: Execute (G, Q_I)
14: **else**
15: Q_U = **Get** Corresponding Update Query by adding "*valid_to*" attribute with value as "*current date/time*"
16: Execute (G, Q_U)
17: **end if**
18: **End**

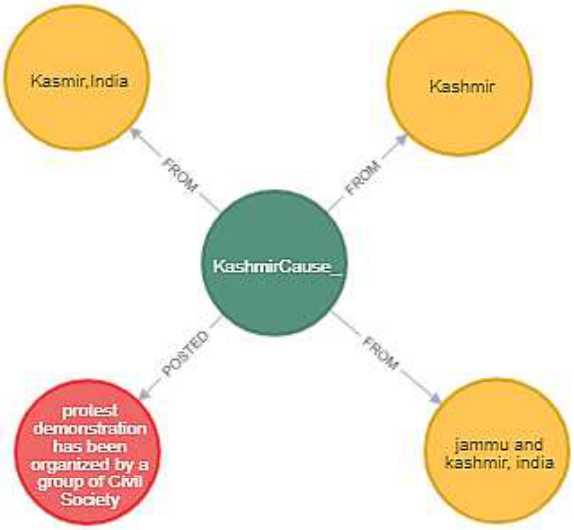
using generated select Query Q_s from parsed results (P) of issued update query (refer to lines 3 to 5). Afterwards, original node is cloned (Updated Cloned Node), but with an updated value of that attribute as per the issued update query, and consequently its "*valid_from*" attribute is set to the "*current date/time*" as a value. Updated cloned node is assigned the same labels, and is associated with all the nodes with same type of edges as per its corresponding original node (refer to lines 6 to 9). Finally, the "*valid_to*" attribute is also added to the original node that stores "*current date/time*" as its value i.e. expire time (refer to line 10). For example, Figure 4.3 shows a snapshot of ZILGDB, where a USER named "KashmirCause_" is from Location "Kashmir" since 15/08/2019, as shown in Figure 4.3a. User updates its location from "Kashmir" to "Kasmir, India". Now, the original node is cloned with all its attributes but the location attribute of cloned node is updated with value "Kasmir, India". The "*valid_to*" attribute of original node is set to the



(a) Before Update



(b) After Update1



(c) After Update2

Figure 4.3: Snapshot of Zero-Information Loss Graph Database

"*current date/time*" and "*valid_from*" attribute of cloned node is set to the "*current date/time*" as a value, as shown in Figure 4.3b. After this, user again updates its location from "Kasmir, India" to "Jammu and Kashmir, india" as shown in Figure 4.3c in the same manner as earlier shown in Figure 4.3b.

As Lucence indexes are attached with all nodes(V) and edges (E) in our database along with timeline which improve their query performance, so search operations with Label Scan algorithms in database run faster and takes O(log(V)) time. In this way, an insert operation takes O(1) time to insert a new node without adding any edge to an existing node. Otherwise, this insert operation takes O(log(V)) time to search a node in the graph and then creates an edge with new node to be created. Similarly, delete operations takes O(log(V)) time to search a node and setting valid_to time as current date/time. Finally, update operation takes O(log(V) time to search a node to be updated and then O(log(V)) time to create a cloned node with edge to exiting nodes with which original node is attached. Hence, update operation takes O(2log(V)) times. Therefore, overall theoretical complexity of algorithm 3 is O(log(V)). However, measuring practical complexity of proposed algorithm is not feasible. In Neo4j database, a Cypher Planner converts an issued Cypher query into an executable query by using a heuristic approach. Hence, a query result is unable to explain that how this information is retrieved from the database.

4.4 Social Data Provenance Framework using ZILGDB: Twitter Case Study

In this section, we present our proposed provenance framework for social media data i.e. Social Data Provenance (SDP) framework on top of ZILGDB. We first design a data model for storing and querying twitter data in Neo4j graph database in an efficient manner.

4.4.1 Graph Data Model for Twitter

Social network is a complex graph-oriented data comprised of various entities with different attributes, and various types of relationships between entities. Social network data can be efficiently modelled using a graph database as it is schema free and flexible, and

every node in a graph may have different number of attributes and may be associated with different number of nodes via explicit relationships. Graph databases natively embrace relationships, and are suitable to store, process, and query social data in an efficient manner irrespective of size of a database. In a graph database, real-world entities are represented as a node. A node can have attributes in the form of a key-value pair and can be associated with other nodes via edges/relationships. Bidirectional relationship between two nodes can be modeled as two separate directed relationships, one in each direction. Nodes and edges with analogues attributes can be grouped under one label.

In the proposed framework, we initially design a data model for Twitter data based on frequently used entities and relationships used in Twitter data analytics. Following entity types i.e. USER, COUNTRY, TWEET, HASHTAG, URL etc., and relationship types i.e. FROM, POSTED, TAGGED, MENTIONED, QUOTED, URL_USED, REPLY_TO etc, are used in Twitter data model design. Like a Labelled Graph, all these entities and relationships are modeled as labeled nodes and edges respectively, and the nodes and edges having similar properties are grouped under one label. Similarly, in accordance with a Property Graph, each of the nodes and edges have their properties/attributes in the form of a key-value pair. The extended graph model is described in the following manner:

- V: Set of Vertices
- E: Set of Edges
- T_N: Set of Vertex Types (TWEET, USER, COUNTRY, HASHTAG, URL)
- f ($T_N \rightarrow V$): Function to assign Type/Label to a Node
- T_E: Set of Edge Type (FROM, POSTED, MENTIONED, QUOTED, TAGGED, URL_USED, REPLY_TO)
- g ($T_E \rightarrow E$): Function to assign Type /Label to an Edge
- A_N: Set of Node Attributes/Properties
- A_E: Set of Edge Attributes/Properties

In accordance with the extended graph model, our Twitter data model is shown in Figure 4.4. It also includes a Timeline *(Year)-[:Has_Month]->(Month)-[:Has_Day]->(Day)-[:HAS_TWEET]->(Tweet)*, where an *Year* is associated with all *Months* of the year using *Has_Month* relationship those are further associated with all *Days* of the month via *Has_Day* Relationship. The *Day* node is associated with the *Tweet* node via *HAS_TWEET* relationship, for all the tweets created on that particular day. Further, each *Day* node is associated to next day using *NEXT* relationship as shown in Figure 4.5. This timeline facilitates querying the graph, especially for range queries [119], such as tweets posted between any two specific days i.e. 16/12/2018 to 18/12/2018 as shown in *Example Query 1* below:

Example Query 1: Retrieve all the tweets posted by a specific user (Screen_Name : KashmirCause_) between 15/12/2018 and 18/12/2018.

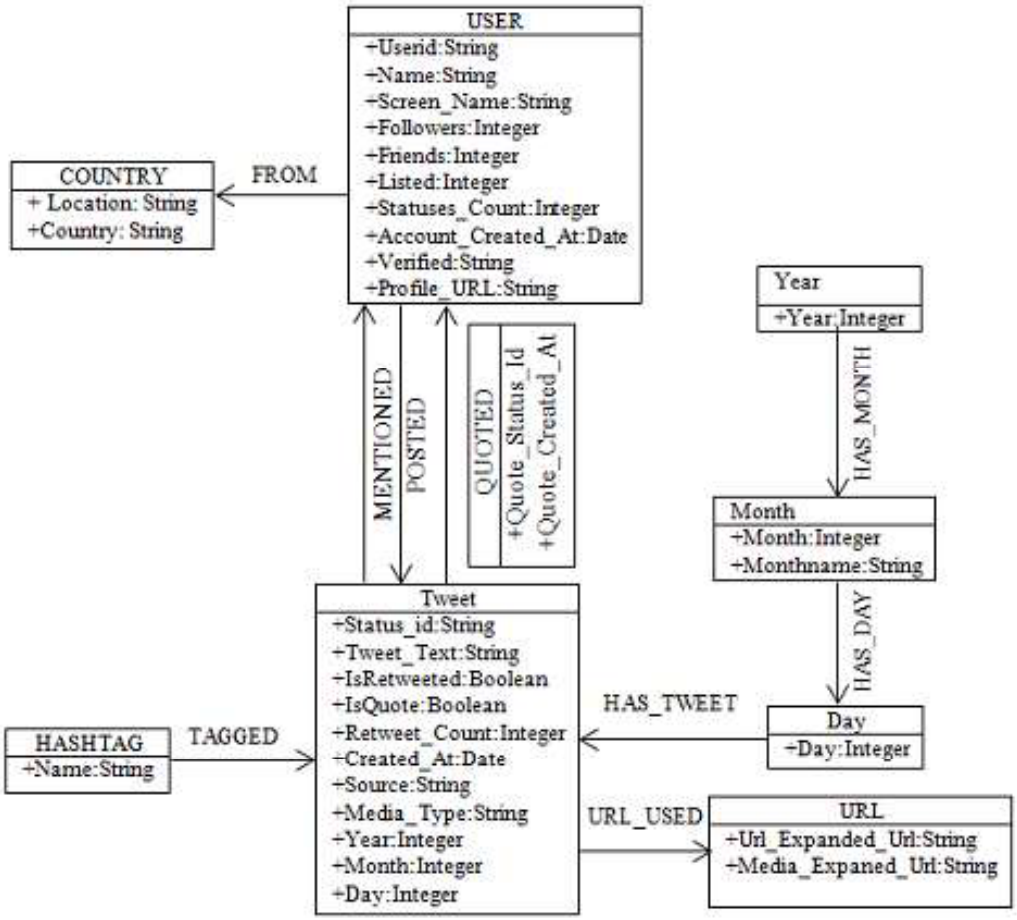


Figure 4.4: Twitter Data Model

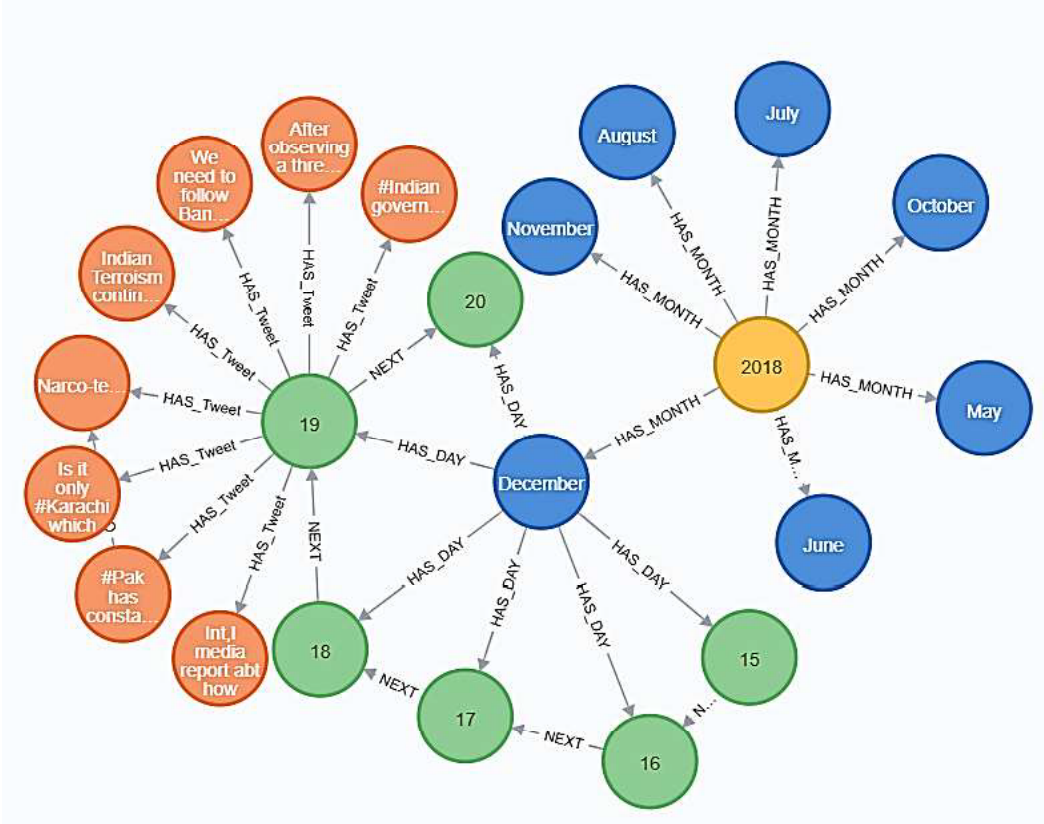


Figure 4.5: Timeline with Tweet

With Timeline: MATCH (y:Year {year: 2018})-[hm:HAS_MONTH]->(m:Month {month: 12})-[hd: HAS_DAY]->(d:Day {day: 15})-[:NEXT*0..3]->(day) with day MATCH (day)-[:HAS_Tweet]->(t: TWEET) <-[:POSTED]-(u:USER {Screen_Name: 'KashmirCause_'}) return *

OR

MATCH start=(y:Year {year: 2018})-[hm:HAS_MONTH]->(m:Month {month: 12})-[hd: HAS_DAY]->(d:Day {day: 15}), end=(y:Year {year: 2018})-[hm:HAS_MONTH]->(m:Month {month: 12})-[hd:HAS_DAY]->(d1:Day {day: 18}), val=(d:Day)-[:NEXT*0..]->(middle)-[:NEXT*0..]->(d1:Day), vals=(middle)-[:HAS_Tweet]->(t:TWEET)<-[:POSTED]-(u:USER{Screen_Name: 'KashmirCause_'}) return *

Without Timeline: MATCH (t:TWEET)<-[:POSTED]-(u:USER {Screen_Name:'Kashmir_Cause_'}) where t.year=2018 and t.month=12 and t.day>=15 and t.day<=18 return *

To retrieve all the tweets posted between two specific days, a user just needs to specify "*start date*" and can traverse all the days up to "*last date*" via NEXT relationship, instead

of searching all the tweets in database. Consequently, all the tweets related to these dates will be retrieved. The Neo4j schema of Twitter data set for efficient queries using timeline is shown in Figure 4.6.

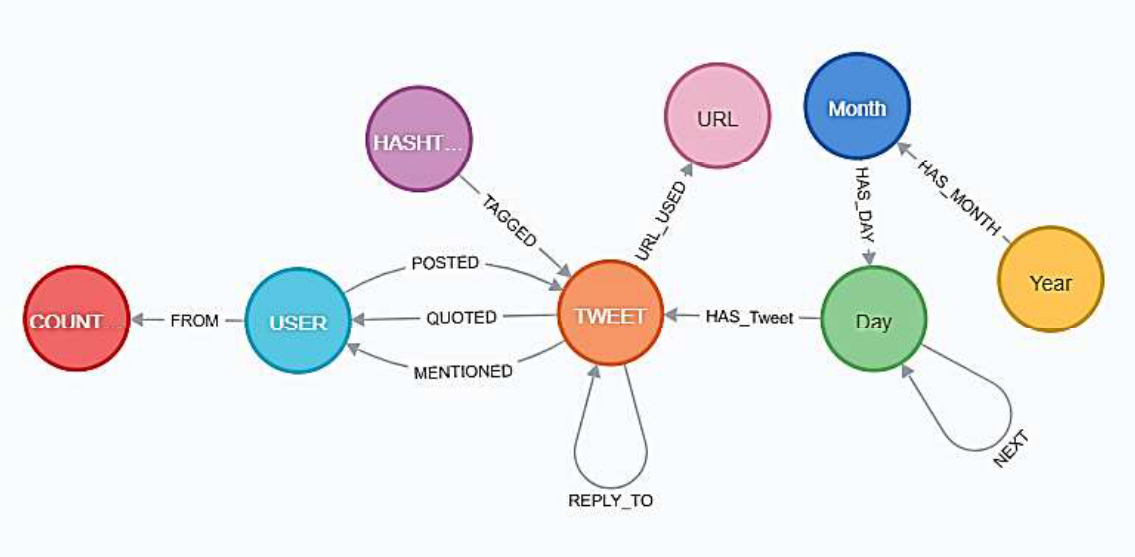


Figure 4.6: Neo4j Schema for Twitter Data Set

4.4.2 Provenance Generation and Storage

Proposed SDP framework supports generation of provenance paths for all result tuples of select, aggregate and historical queries. The captured provenance path is also stored in Provenance Graph Database (PGDB) for visualisations and analysis. We state the problem of provenance path generation as below:

Problem 2: *Provenance Paths Generation Problem.*

Given a Graph G(V,E) and a cypher query (Q), here our objectives are:

1. To generate a query graph G_Q, by executing Q on G, where G_Q is a subgraph of G that comprise of all source nodes contributed towards generation of all result tuples T_n.

2. For every result tuple $t \in T_n$, perform DFS on query graph G_Q to generate provenance paths P of t and insert it in final provenance graph $G_p(V_p,E_p)$.

Flow diagram for Provenance Graph Database Construction (refer to Figure 4.7) shows the steps starting from data model designing as explained in previous section. As per

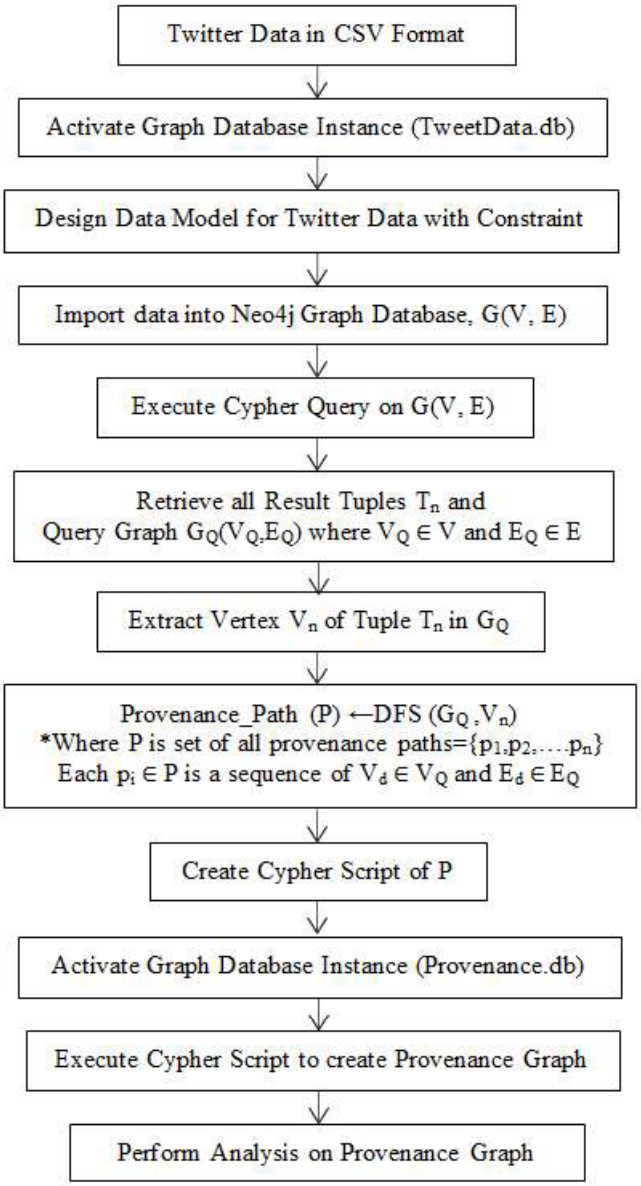


Figure 4.7: Provenance Graph Construction Flow Diagram

the data model, scripts are written for importing the data from csv file into Neo4j and Graph Database G(V,E) is constructed. Then, for any Cypher Query executing on Graph G(V,E), provenance paths of every result tuple of query is generated in the form of cypher script. Finally, cypher script of provenance path is executed to generate provenance graph database.

Algorithm 4 shows high-level details of provenance graph generation that accepts graph *G(V,E)* and *Cypher Query Q (Select or Aggregate)* as inputs and returns provenance graph of all result tuples T of Q as output. Initially, the cypher query Q is executed on

Algorithm 4 ***ProvGraph Generation***: Extraction of Provenance Graph for a Result Tuple of Query Q on Graph G

Input: Data Graph G(V, E) and Cypher Query Q

Output: Provenance Graph $G_P(V_P , E_P)$ of Query Q on Graph G where $V_P \in V$ and $E_P \in E$

1: **Obtain** all Result Tuples T_n, Query Graph $G_Q(V_Q,E_Q) \leftarrow$ Execute (G,Q)
//Where $V_Q \in V$ and $E_Q \in E$ and For each $T_i \in T_n$, $T_i = (A_1, A_2, \ldots, A_n)$
//Where A_i = Property of a Node or an Aggregate Function
2: **for all** $T_i \in T_n$ **do**
3: Select A_i *// A_i should be Property of a Node not an Aggregate Function*
4: $V_t \leftarrow$ Extract Vertex V_i of A_i in G_Q
5: Provenance_Path (P) $\leftarrow$ DFS (G_Q ,V_t)
//Apply DFS on V_t, P is set of all provenance paths=$p_1,p_2,\ldots,p_n$, Each $p_i \in P$ is
//a sequence of $V_d \in V_Q$ and $E_d \in E_Q$
6: $G_P(V_P , E_P) \leftarrow$ Insert P *//G_P is Provenance Graph*
7: **end for**
8: Return $G_P(V_P , E_P)$

Graph G which returns all the result tuples as well as *Query Graph (G_Q)* containing all the nodes and associated edges contributed to all result tuples of Q, as complete provenance graph of Q (refer to line 1). After that, for some/all result tuples, corresponding vertex is extracted from *Query Graph (G_Q)* (refer to lines 2 to 4).Then, DFS is applied on extracted vertex to generate provenance paths of result tuple (refer to line 5). Provenance paths are then added to Provenance Graph Database (PGDB) for further analysis i.e. provenance visualization (refer to line 6). As far as the complexity of proposed algorithm in concerned, select query Q takes O(log(V)) time to generate Result Tuples Tn and Query Graph $G_Q(V_Q, E_Q)$. It takes n * $O(\log(V_Q))$ + $O(V_Q + E_Q)$ times to find the vertex of each result tuple and to perform DFS on that sub graph. Therefore, overall theoretical complexity of the algorithm 4 is O(log(V)) + n*$O(\log(V_Q))$ + $O(V_Q+E_Q)$. But, practical complexity of proposed algorithm, is not feasible to measure as explained earlier for Algorithm 3. Illustrative example queries of proposed algorithm are presented below:

Example Query 2: Find top 50 most retweeted tweets by any user from a specific location "Kashmir" in a given range of days with the retweet count, tweet text, user's name, in descending order of the retweet count.

Cypher Query 2: MATCH (c:COUNTRY)<-[f:FROM]-(u:USER)-[p:POSTED]->(t:TWEET)<-[ht:HAS_Tweet]-(d:Day)<-[hd:HAS_DAY] -(m:Month)<-[hm:HAS_MONTH]-(y:Year) where c.Location ='Kashmir' and y.year=2018 and m.month=12 and d.day>=18 and d.day<=20

return t.Retweet_Count AS retweet_count, t.Tweet_Text as tweet, u.Screen_Name as user_screenname, c.Location as user_location order by t.Retweet_Count desc limit 50

Snapshot of Example Query 2 result is shown in Figure 4.8. Here, first row shows most retweeted tweet posted by user named "KashmirCause_" from location "Kashmir" and so on for other rows in descending order of retweet_count. To explore why this result has appeared in result set, and how it is derived; we need to know about provenance information for the query result set. Figure 4.9 shows partial provenance graph of all result tuples (labeled as '*QUERYTUPLE*') i.e. Q1t1, Q1t2 etc. of query Q1 (labeled as '*QUERY*'). Further, each result tuple node is associated with all source nodes contributed towards it via '*provenance*' relationship. Figure 4.10 shows provenance graph of result tuple 1 i.e. Q1t1 of Q1. Provenance graph also stores the complete information about query which has been executed i.e. issued query, time of execution as the attributes of '*QUERY*' node.

"retweet_count"	"tweet"	"user_screenname"	"user_location"
34	"proper demonstration has been organized be a group of Civil Society Organization including the People's Union for Civil Liberties, National Alliance of People's Movements Protests to be Held At Janter Manter Delhi Against Pulwama Killings. #Kashmir https://t.co/viUPL5MBBjO"	"KashmirCause_"	"Kashmir"
32	"Video: Founders National Federation for Christian Woman Candlelight Protest at Janter Manger in Delhi against Civilian killings at #Pulwama South #Kashmir https://t.co/4cFPwGwSSt"	"KashmirCause_"	"Kashmir"
23	"Indonesian widow of slain Pulwama man leaving Kashmir on a heartbroken note – #Kashmir #Conflict #Pulwama #CivilanKillings https://t.co/C56HxVM8Lo"	"FreePressK"	"Kashmir"

Figure 4.8: Snapshot of Result of Example Query 2

Example Query 3: Find top N users those have used a Hashtag "India" in a tweet in December 2018 with the most number of retweets, user's name, tweet text, and retweet count in descending order of the retweet count.

Cypher Query 3: MATCH (y:Year{Year:2018})-[hm:Has_Month]->(m:Month{Month:12})-[hd: Has_Day]->(d:Day)-[ht:HAS_ TWEET]->(t:Tweet) with t MATCH (u:USER)-[p:POSTED]->(t)<-[tg: TAGGED]-(h:HASHTAG) where h.Hashtagname = 'India' return distinct u. Screen_Name as user_screenname, t.Retweet_Count as retweet_count, t.Tweet_Text as

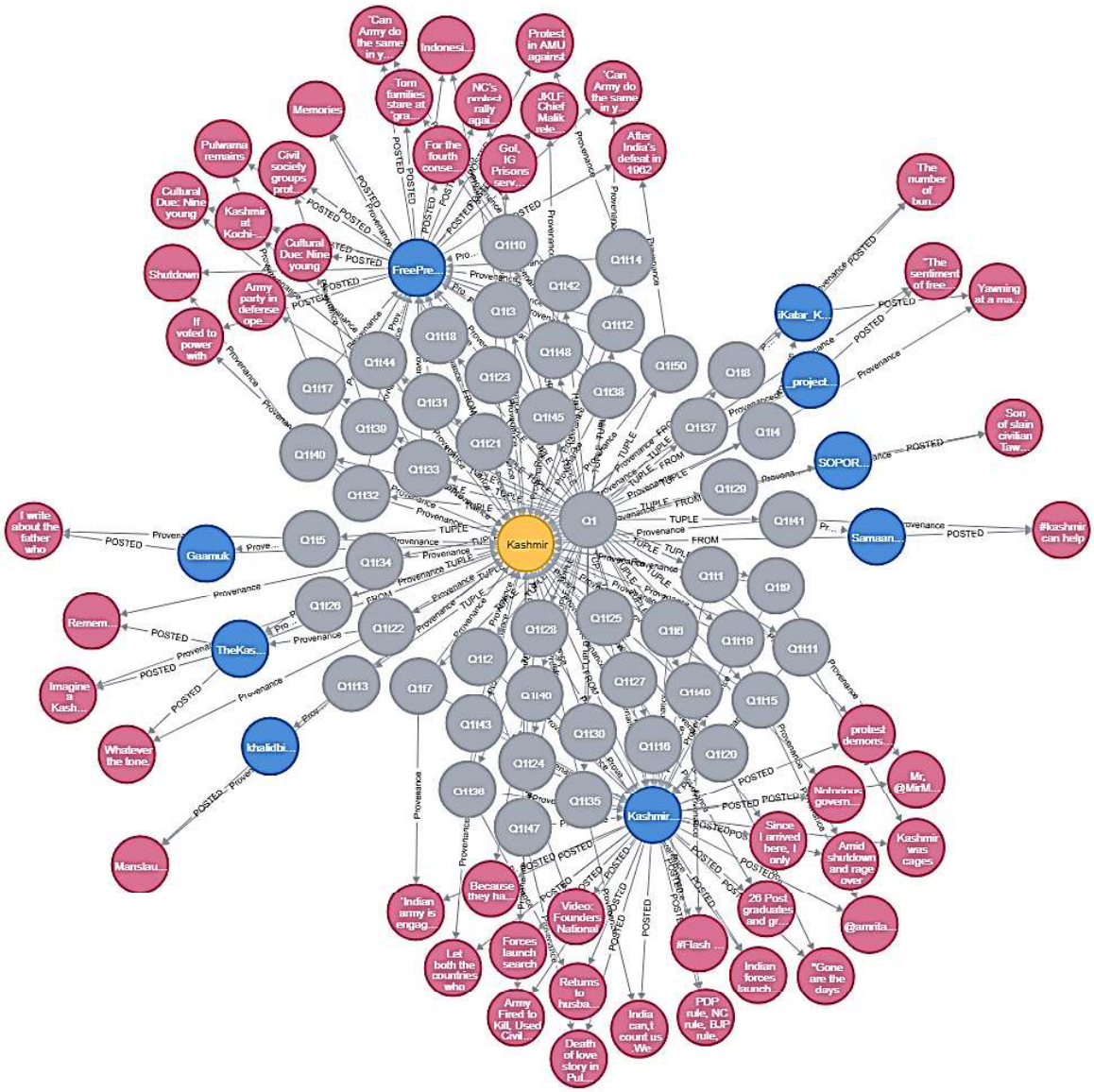


Figure 4.10: Provenance of Result Tuple t1 of Example Query 2

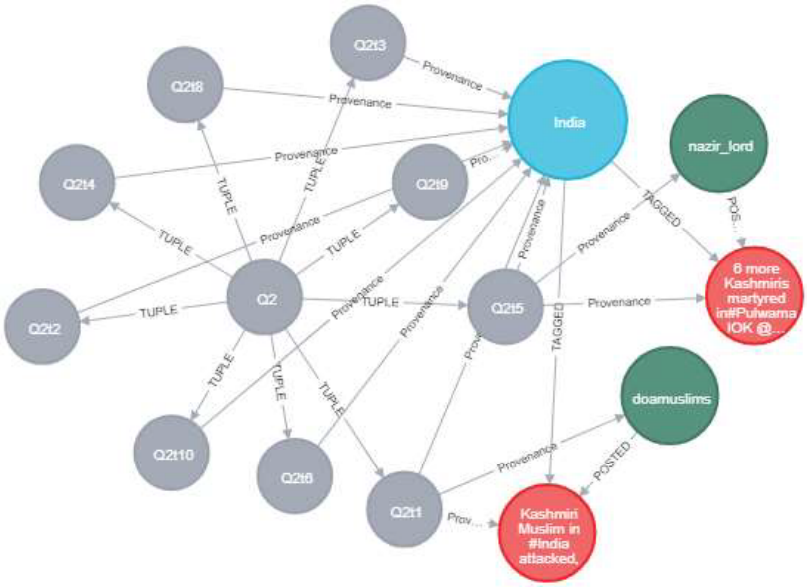


Figure 4.11: Partial Provenance Graph of Example Query 3

tweet order by retweet_count desc limit 100

Partial provenance graph of above query is shown in Figure 4.11.

Example Query 4: Find topmost hashtag which appeared in most number of countries along with the number and list of the distinct countries it appeared in.

Cypher Query 4: MATCH (h:HASHTAG)-[tg:TAGGED]->(t:TWEET)<-[p:POSTED]-(u:USER)-[f:FROM]->(c:COUNTRY) return (count(distinct(c.Country))) AS Longest_Path, collect(distinct (c.Country)) as Country, h.Hashtagname AS Hashtag limit 1

The proposed framework is also capable to capture provenance for aggregate queries, as given in Example Query 4. The query result of this aggregate query describes that users from 53 different countries are using a common most popular hashtag "Kashmir"

Longest_Path	Country	Hashtag
53	[Afghanistan, Armenia, Australia, Azad Jammu and Kashmir, Bahrain, Bangladesh, Belgium, Canada, Chile, China, Egypt, England, Finland, France, Germany, Hungary, Iceland, India, Indonesia, Iran, Iraq, Ireland, Israel, Italy, Jammu and Kashmir, Japan, Korea, Kuwait, Malaysia, Nepal, Netharlands, New Zealand, Nigeria, Norway, Pakistan, Palestine, Portugal, Qatar, Russia, Saudi Arabia, Singapore, South Africa, Spain, Sri Lanka, Swedan, Switzerland, Syria, Thailand, Turkey, UK, United Arab Emirates, USA, Yorkshire]	Kashmir

Figure 4.12: Snapshot of Result of Example Query 4

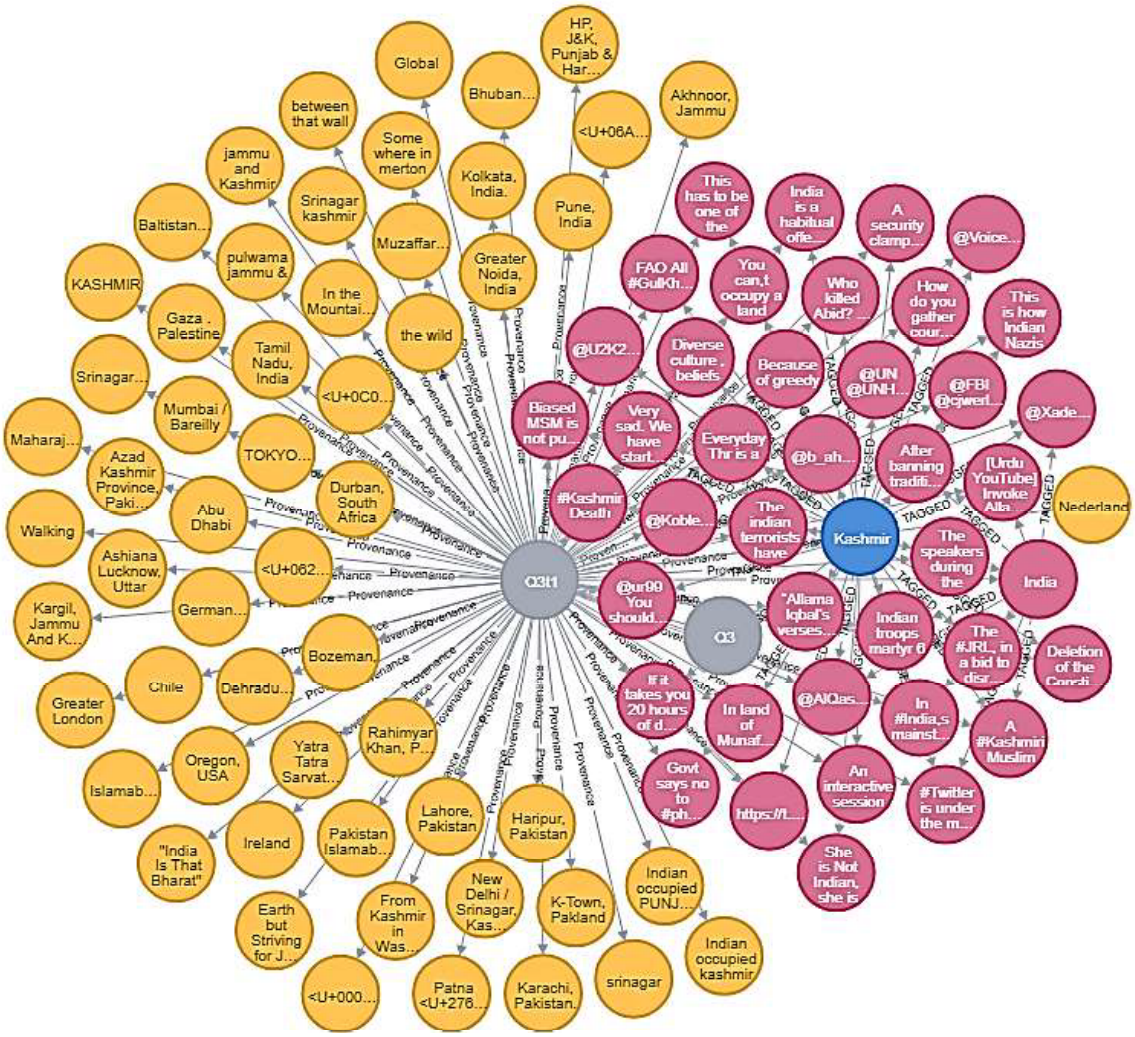


Figure 4.13: Snapshot of Provenance of Example Query 4

in their posted tweets as shown in Figure 4.12. The provenance graph of this aggregate query is shown in Figure 4.13, which can be analysed further for justifying query results such as why does the country name "Australia" is appeared in the result set? how many users from this country are using the most popular hashtag in their posted tweets? etc.

4.4.3 Querying Provenance

Provenance of a query result provides the details of all the relevant paths originating from the source graph which contributed to generate the query result as explained in previous section 4.4.2. Querying provenance information aids in explaining and justifying the result of a data retrieval query through forward or backward tracing. It is helpful in social data analytics where the accountability of an analysis is largely relying on the data quality

and trustworthiness of input data. Also, it has the ability to trace out the root cause in a social data analytics. Provenance enabled social media can offer an effective approach to address increasingly challenging issue of trust in social media. We state the querying provenance problem as follows:

Problem 3: *Querying provenance for a specific purpose.*

Consider a Provenance Graph $G_p(V_p,E_p)$, our objectives are:

1. To find a subgraph $G_{ps}(v_{ps},e_{ps})$ of graph G_p such that $v_{ps} \in V_p$, $e_{ps} \in E_p$ which is contributing directly or indirectly to a specific result tuple for various purposes like audit trail, tracing the source of a result tuple etc.

2. To extend the cypher query by including following user-defined constructs viz. "*all*", "*instance*", "*valid_on now*", and "*valid_on 'date'*" for historical data queries.

In this section, we explain the querying provenance from both perspectives as mentioned in above stated problem through some example queries.

Example Provenance Query PQ1: Let's consider the result set and partial provenance graph of Example Query 2 shown in Figures 4.8 on page no. 115 and 4.9 on page no. 116 respectively. We now explain the derivation of result tuple t2 in the result set.

Cypher Query PQ1: *MATCH (n:QUERY)-[t:TUPLE]->(n1:QUERYTUPLE)-[p:Provenance]->(n2) where n.qid='Q1' and n1.qtid='Q1t2' RETURN n2*

After executing the above cypher query PQ1, we obtain the provenance path of result tuple t2 as shown in Figure 4.14. This provenance path shows the three nodes from source graph contributing towards generation of t2. Relationships among these nodes explain '*How*' they are contributing towards generation of result tuple.

Figure 4.14: Result of Provenance Query PQ1

Proposed framework also supports multi-depth provenance querying to know about both the direct and indirect sources of a result tuple. Multi-depth provenance query upto depth 2 is explained by following example provenance query PQ2.

Example Provenance Query PQ2: Let's consider the result tuple with tuple id Q1t2 shown in Figure 4.9 on page no. 116, find the direct and indirect sources of Q1t2 upto depth 2.

Cypher Query PQ2: MATCH (n:QUERY)-[t:TUPLE]->(n1:QUERYTUPLE)-[*..2]->(a) where n.qid= 'Q1' and n1.qtid= 'Q1t2' return *

Proposed framework also allows querying historical data by introducing four new query constructs viz. *"instance"*, *"all"*, *"valid_on now"*, and *"valid_on 'date'"* as an extended Cypher query constructs. It supports querying a data element with a given time in the past and with a time range specified in the query statement. Initially, the user issues a provenance query with extended cypher query constructs which is automatically rewritten and passed to ZILGDB for further execution. The example provenance queries (PQ3, PQ4, PQ5, and PQ6) for querying historical data are explained below:

Example Provenance Query PQ3: Display the current location of user for the graph shown in Figure 4.3 on page no. 107.

Extended Cypher Query PQ3: MATCH instance (u:USER)-[:FROM]->(c:COUNTRY) where n.screen_name= 'KashmirCause_' return c.Location valid_on now

Rewritten Cypher Query PQ3: MATCH (n:USER) -[:FROM]->(c:COUNTRY) where n. screen_name= 'KashmirCause_' and c.valid_to is null return c.Location

Above query generates current location i.e. "Jammu and Kashmir, india" in its result set.

Example Provenance Query PQ4: Display the location of user on 01/10/2019 for the graph shown in Figure 4.3 on page no. 107.

Extended Cypher Query PQ4: MATCH instance (u:USER)-[:FROM]->(c:COUNTRY) where n. screen_name= 'KashmirCause_' return c.Location valid_on date('2019-10-01')

Rewritten Cypher Query PQ4: MATCH (n:USER) -[:FROM]->(c:COUNTRY) where n. screen_name= 'KashmirCause_', date('2019-10-01')>=c.valid_from and c.valid_to>=date

('2019-10-01') return c.Location

Above query generates location i.e. "Kashmir" in its result set.

Example Provenance Query PQ5: Display all the location updates of user till now for the graph shown in Figure 4.3 on page no. 107.

Extended Cypher Query PQ5: MATCH all (u:USER)-[:FROM]->(c:COUNTRY) where n. screen_name='KashmirCause_' return c.Location valid_on now

Rewritten Cypher Query PQ5: MATCH (n:USER) -[:FROM]->(c:COUNTRY) where n. screen_name= 'KashmirCause_' return c.Location

Above query generates locations i.e. "Kashmir", "Kasmir, India", and "Jammu and Kashmir, india" in its result set.

Example Provenance Query PQ6: Display all the location updates of user valid on 20/11/2019 for the graph shown in Figure 4.3 on page no. 107.

Extended Cypher Query PQ6: MATCH all (u:USER)-[:FROM]->(c:COUNTRY) where n. screen_name='KashmirCause_' return c.Location valid_on date('2019-11-20')

Rewritten Cypher Query PQ6: MATCH (n:USER) -[:FROM]->(c:COUNTRY) where n. screen_name ='KashmirCause_' and (date('2019-11-20')>=c.valid_to or (c.valid_from<= date('2019-11-20') and c.valid_to is null)) return c.Location

Above query generates all the locations of user till 20/11/2019 i.e. "Kashmir" and "Kasmir, India" in its result set.

4.5 Experimental Setup and Results

4.5.1 Experimental Setup

All the experiments are performed on a Windows machine with Intel i7-8700 processor @ 3.20GHz and 16GB RAM. Neo4j Desktop version 1.2.1 with embedded Neo4j enterprise edition 3.5.6 has been used as a Graph Database. Neo4j extension library APOC (Awesome Procedures On Cypher) version 3.5.0.4 is included to call APOC library procedures with some modifications to import/export the social data set from csv files to Neo4j graph

database and vice versa. Java version 8 has been used as front end programming language to interact with Neo4j graph database. To perform experimental analysis, publicly available twitter dataset related to an incidence of 11 killings at Pulwama district of Jammu and Kashmir, India on 15/12/2018 [158], has been used. By using cypher scripts, experimental dataset is fed into Neo4j graph database, and then ZILGDB is designed as per the data model in section 4.4.1. Our initial database consists of around 35000 nodes and 78000 edges, and grows gradually after performing each data update operation. Provenance graph database (PGDB) is being created to store the captured provenance information for each query executed on ZILGDB. We now present the analysis of proposed framework based on execution overhead in terms of provenance capturing and querying.

4.5.2 Results and Discussions

Analysis of framework is performed on following parameters viz. provenance capture overhead, and performance of querying provenance information.

4.5.2.1 Provenance Capture Analysis

To perform experimental analysis of provenance capture, we have executed 24 different data retrieval queries on ZILGDB to capture their provenance information using timeline wherever it is required. A sample set of data retrieval queries with their usefulness and required parameters are shown in Table 4.1. As we have already explained in section 4.4.1, through Example Query 1, that the performance of a range query using timeline is much efficient as compared to a query executed without timeline (refer to Figure 4.15 on page no. 124). In a range query, the start and end dates are mentioned explicitly in the query statement, therefore with timeline execution, all the tweets those are posted in-between these dates (including both the dates) and linked through "HAS_Tweet" relationships are only required to search instead of searching the whole database. Thus, no any significant change is measured in the query performance after increasing the size of database. But, in the case of without timeline execution, we have to search whole database to retrieve all the tweets posted between these two dates. This degrades the performance, with increase in the number of tweets.

Table 4.1: Sample Data Retrieval Queries with usefulness

Query No.	Query
Q1	Find top N most retweeted tweets by the users from a given location in a given range of days with the retweet count, tweet text, user's name, in descending order of the retweet count.
	Parameter: N=50, Location= "Kashmir", days are 18th December to 20th December 2018 **Usefulness:** This query finds 50 most influential tweets in the given days of a month and the user who posted them.
Q2	Find top N users who used a Hashtag "India" in a tweet with the most number of retweets with user's name, tweet text, and retweet count in descending order of the retweet count.
	Parameter: N=100, Hashtag="India". **Usefulness:** This query finds top 100 influential users who used a given hashtag that may represent a certain agenda.
Q3	Find top N hashtags that appeared in the most number of Countries with the number of countries it appeared in along with the list of the distinct countries it appeared.
	Parameter: N=1 **Usefulness:** This query finds top 1 hashtag which is most widely spread across Countries all over the world, which could indicate a certain agenda which is discussed at global level.
Q4	Find distinct locations along with the month and the date of a tweet posted with a given hashtag in a given range of days, with tweet text and retweet count in descending order of the retweet count.
	Parameter: Hashtag="Kashmir", days are 1st December to 31st December 2018 **Usefulness:** This query finds the location of most retweeted tweet with a given hashtag which could indicate a certain agenda that is widely discussed.
Q5	Find N users who used a given set of hashtags in their tweets with the user's name and the country to which the user belongs in the alphabetical order of the names.
	Parameter: N=100, Hashtag in ["Kashmir","kashmir","Pulwama"] **Usefulness:** This query finds top 100 users who share similar interests (based on hashtags).
Q6	Find the users from a given Country who used a given hashtag in tweets posted in a given range of days along with the count of tweets posted with that hashtag order of the tweet counts.
	Parameters: Hashtag="Pulwama", Country="USA", days are 15th December 2018 to 17th December 2018. **Usefulness:** This query finds the users who used a given hashtag most often in a given Country. These users could be trying to influence an agenda within the Country.

Query No.	Query
Q7	Find top N tweets of a given day posted by a given user in a specific country tagging a given hashtag along with the tweet text and the user's screen name and location.
	Parameters: N=1000, Name:'Dilip Kumar paliwal', Hashtag="Kashmir", Country="India", days are 1st December 2018 to 31st December 2018. **Usefulness:** This query will help to know more about the content of the tweets with the hashtag of interest by any given user of a given country.
Q8	Find top N most followed users with the user's name, the number of followers, and location in descending order of the number of followers.
	Parameters: N=10 **Usefulness:** This query finds the most influential user in terms of the number of Followers.
Q9	Find the list of distinct hashtags that appeared in one of the country in a given list in a given day with the list of the hashtags and the country in which they appeared.
	Parameters: N=10, Country in ['India', 'Pakistan', 'UAE'], days are 1st December 2018 to 31st December 2018. **Usefulness:** This query will find common interest among the users in the countries of interest.
Q10	Find top N tweets with hashtags posted by a user of a specific location on a given day with the tweet text, the hashtag, the user's Screen_name of the user who posted the tweet.
	Parameters: N=10, Location="USA" , days are 1st December 2018 to 31st December 2018. **Usefulness:** This query allows to find the context in which the hashtags were used.

Initially, all the data retrieval queries are executed 42 times without any provenance capturing mechanism. Afterwards, the same set of data retrieval queries are executed 42 times with provenance capturing mechanism. To calculate the average execution time of each query, we have dropped the maximum and the minimum execution time, and then

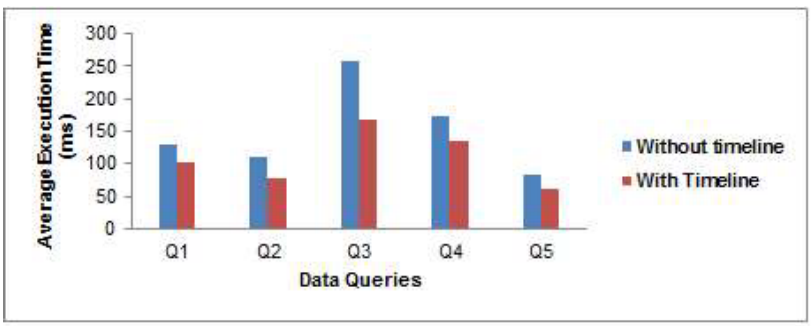


Figure 4.15: Query Performance with and without timeline

taken average of the remaining 40 values. Average execution times of all the queries are given in milliseconds (ms). Because of the couple of order difference in execution times, the execution performances of all the queries without provenance capture and with provenance capture mechanism are shown separately in Figures 4.16 and 4.17, respectively. We have found that the select queries which are generating larger number of result tuples are taking more time for capturing and storing provenance information as compared to those select queries having lesser number of results tuples in their result set. This is because the proposed framework capture and store provenance information of each result tuple of a query, which increases the execution time with increase in number of result tuples. It can be seen in Figure 4.17 that Queries Q2 and Q23 are taking longer execution time as compared to other queries, because the number of result tuples generated by each of them is 100.

The proposed framework also provides provenance capturing support for aggregate queries using aggregate functions such as collect, count, min, and max. Here, query Q3 (an aggregate query) uses collect function to retrieve all the countries which are using

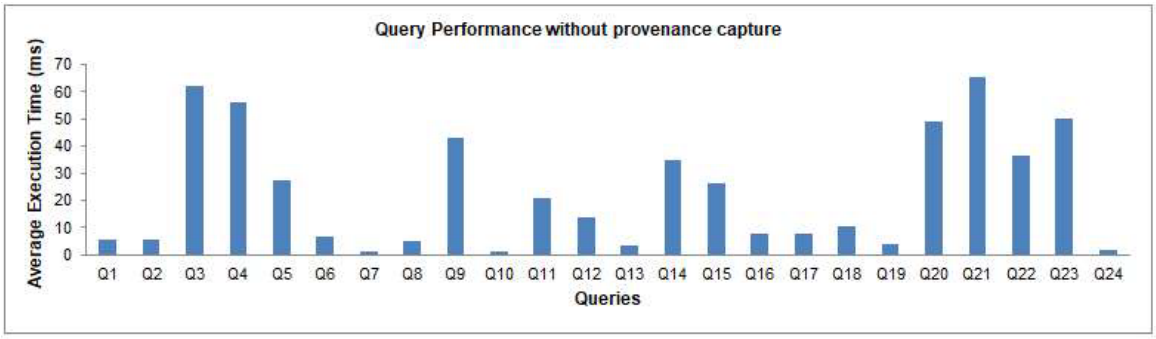


Figure 4.16: Query Performance without Provenance Capture

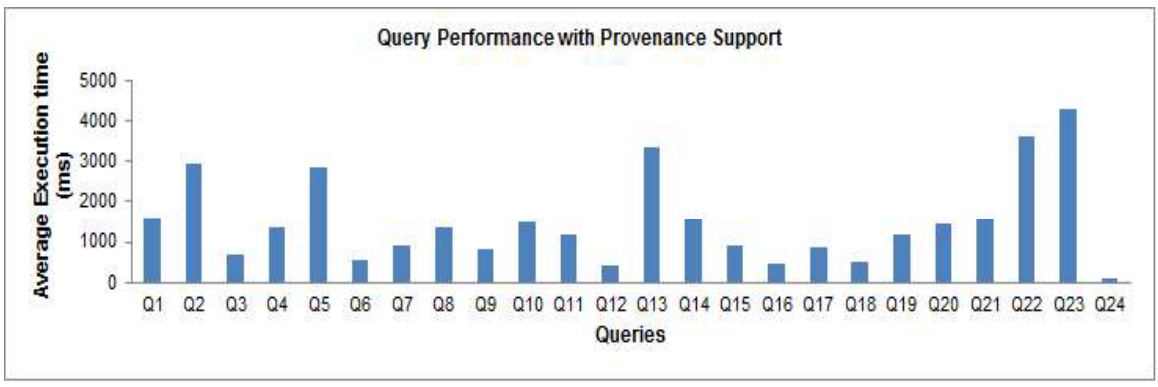


Figure 4.17: Query Performance with Provenance Capture

topmost hashtag in their tweets and returns only one result tuple in its result set. On the other hand, query Q6, shown in Figure 4.17, is not an aggregate query like Q3 and generates 6 result tuples in its result set, still its performance is better than Q3, even though it generates more number of result tuples as compared to Q3. On the other hand, query Q9, shown in Figure 4.17, is also an aggregate query generating 10 result tuples and its execution time is comparatively higher than aggregate query Q3 which is generating 1 result tuple. Thus, it can be concluded that aggregate queries are taking longer execution time as compared to select queries. Secondly, execution time of aggregate queries with more number of result tuples is comparatively higher than other aggregate queries with lesser number of result tuples.

The proposed framework also supports provenance capturing for insert, delete, and update queries. A sample set of data update queries are shown in Table 4.2. The following parameters are used to capture the provenance information for update queries, viz. "*Previous value before update*", "*time till the previous value was valid*", and "*time from when new updated value is valid*" as shown in sample graph in Figure 4.3. Similarly "*valid_from*" and "*valid_to*" parameters are used in the case of insert and delete query respectively. The provenance information captured for insert, delete, and update queries can be used for both historical data queries, and queries executed in the past (historical query) on a specific time.

Query No.	Query
Q1	Update location of user with screen name "KashmirCause_".
Q2	Update name of user with screen name "KashmirCause_".
Q3	Update URL of a user with given screen name.
Q4	Insert new post of a user.
Q5	Delete a specific post of a user.

Table 4.2: Sample Data Update Queries

4.5.2.2 Provenance Querying Analysis

The performance analysis of querying provenance information stored in Provenance Graph Database (PGDB) is presented in this section. A set of 20 provenance queries are executed for analysis of querying provenance. Out of these 20 queries, a sample set of 10

Query No.	Query
Q1	Why Hashtag "Kashmir" has appeared in the result set of Query Q3?
Q2	Retrieve all the nodes which have contributed to produce tuple t1 of Query Q1.
Q3	Why Hashtag "Kashmir" has arrived as longest path of countries in Query Q3?
Q4	How Tweet t1 has been derived as result in Query Q7?
Q5	Which users from "Jammu and Kashmir" uses top hashtag(Maximum Tagged) in Query Q3?
Q6	How and which user has contributed to produce result tuple t2 of Query Q2?
Q7	Display the current location of user with screen name "KashmirCause_".
Q8	Display the location of user with screen name "KashmirCause_" on 01/10/2019.
Q9	Display all the location updates of user with screen name "KashmirCause_" till now.
Q10	Display all the location updates of user with screen name "KashmirCause_" till 11/10/2019.

Table 4.3: Sample Queries on Provenance

provenance queries are shown in Table 4.3. This sample set contains both the type of provenance queries i.e. provenance queries for query results, and provenance queries for historical data. Provenance Graph Database consists of approximate 22000 nodes and 43000 relationships at the time of execution of these provenance queries. Each query in the provenance query set is executed 12 times. To calculate the average execution time of each query, we have dropped the maximum and the minimum execution time, and then taken average of the remaining 10 values. A performance analysis of 10 provenance queries for query results are shown in Figure 4.18. The average execution times of all the queries are mentioned in milliseconds (ms).

As we have seen in the section 4.4.2 that the provenance graph of a query result contains a separate node for each data retrieval query, labelled as "*QUERY*", and their result tuples, labelled as "*QUERYTUPLE*". These "*QUERYTUPLE*" nodes are associated with corresponding "*QUERY*" node via "*TUPLE*" relationship. "*QUERYTUPLE*" nodes are further linked with all the source nodes those are contributing to produce it through a relationship, labelled as "*provenance*" (refer to Figure 4.9 and Figure 4.10 on page no. 116). This makes the searching process faster by reducing the search space to a particular portion of the provenance graph rather than the whole graph. As a result, a small execution time is

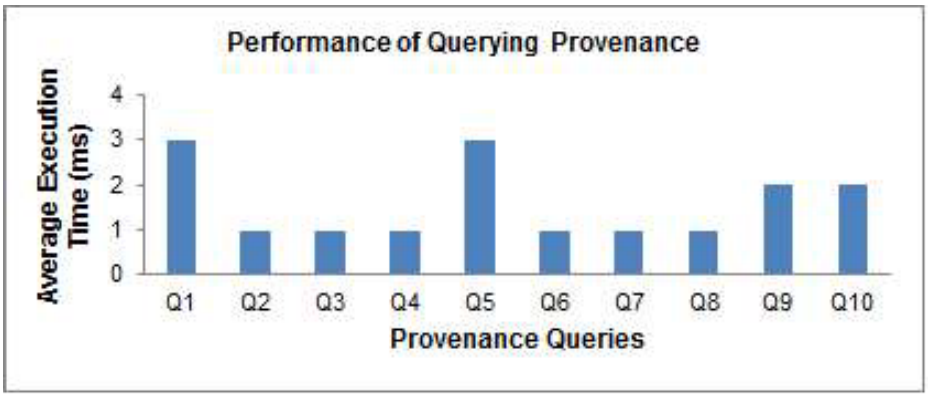


Figure 4.18: Querying Provenance for Justifying Result Tuples

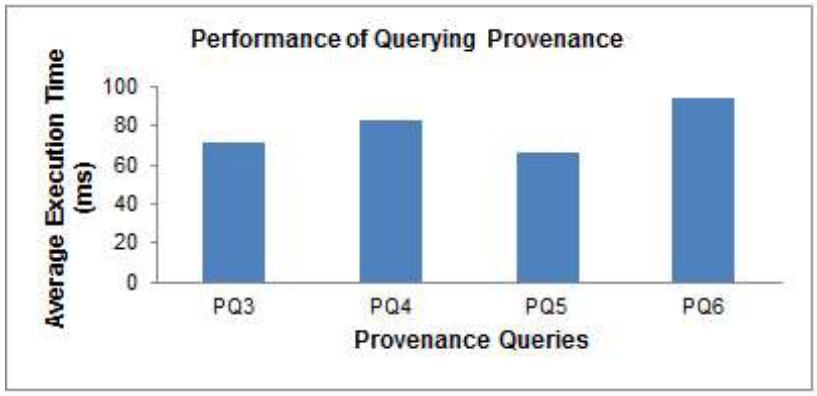


Figure 4.19: Querying Provenance for Historical Data

measured in querying provenance for query results. In this way, it is clear from Figure 4.18 that the performance of querying provenance for query results is very efficient.

In case of querying provenance for historical data queries, such as PQ3, PQ4, PQ5, and PQ6 (explained in section 4.4.3), a longer execution time is measured as compared to querying provenance for query results (refer to Figure 4.19). As the query statement is initially written in the extended cypher query using various user-defined constructs such as "*all*", "*instance*", "*valid_on now*", and "*valid_on 'date'*", which is further required to rewrite the statement into the Neo4j cypher query for execution on database. This rewriting process may incur some execution overheads.

4.6 Application Scenario

In this section, applicability of proposed social provenance framework is presented for investigating terrorist attack, and identifying suspects, and their associated communities

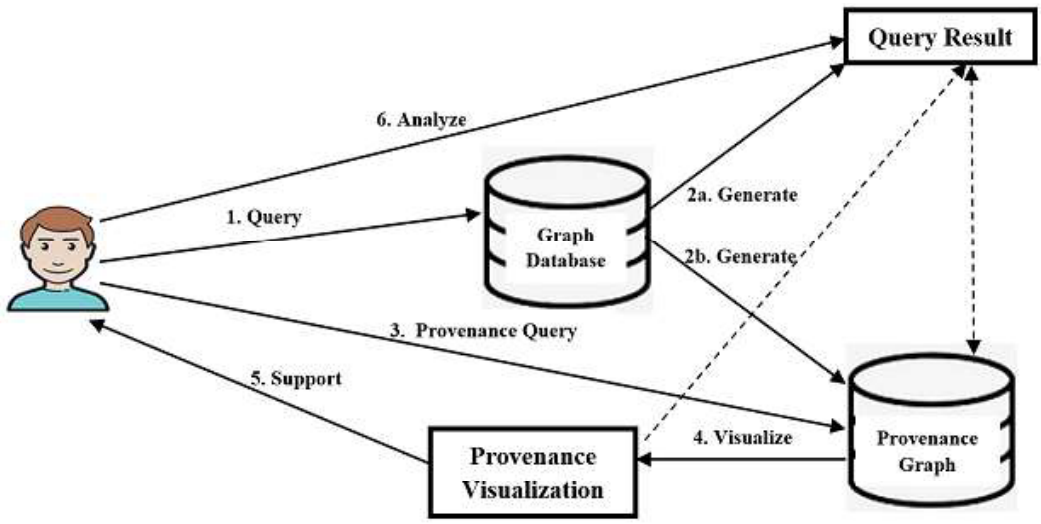


Figure 4.20: A Cyclic Process Model Based on Provenance Framework

on social media especially in Twitter's network. A cyclic process model based on proposed framework is shown in Figure 4.20. Initially, a large volume of twitter dataset is fed into Neo4j graph database for performing further analysis based on suspicious tweets and hashtags. Whenever a user issues a query on database with specific predicates, a set of result tuples, based on issued query are generated. By employing the proposed provenance framework, a set of provenance graphs for all the result tuples are also generated, which consists of a set of nodes of the source graph, which are contributing to produce it. Now, these provenance graphs are required to be visualized and analyzed to obtain *Who*, *Why*, and *How* Provenance of a suspicious tweet exists in the result set. Further, all the other tweets posted by a suspicious user can be retrieved from the source database using proposed framework. This provenance data can be visualized in a way that may provide support back to the on-going investigation process. In this way, a user can visualize the provenance graph to identify such suspicious person and his linked communities.

A use case scenario is given below to explain the use of above process model in terrorist attack investigation by identifying suspicious person and his linked communities in Twitter's Network. A publicly available twitter data set related to an incidence of 11 killings at Pulwama district of Jammu and Kashmir on 15/12/2018 are used for the analysis purpose. The data set consists of around 10,000 tweets those are reactions to that incidence. Using cypher scripts, data set is first modelled into Neo4j graph database as per the data model explained in section 4.4.1. Now, by employing the proposed prove-

nance framework, our goal is to identify the suspects those are not in favour of Indian Army. Therefore, we have executed the following query to retrieve all the tweets those are posted on 15/12/2018 with a hashtag "IndianArmy".

Example Query 5: Find all the tweets posted on 15/12/2018 in which hashtag "IndianArmy" is used.

Cypher Query 5: MATCH (y.YEAR)-[:Has_Month]->(m:MONTH)-[:Has_Day]->(d:DAY)-[:HAS_ TWEET]->(t:TWEET)<-[tg:TAGGED]-(h:HASHTAG) where h.Hashtagname = 'IndianArmy' and t.year=2018 and t.month=12 and t.day=15 return t.Tweet_Text as TWEET limit 100

In response to the above query execution, 53 result tuples and their provenance graphs are generated. After analyzing all the result tuples, suspicious or provoked tweets exist in the result set are marked. A snapshot of partial result set of this query execution is shown in the Figure 4.21, in which a highlighted tweet is looking suspicious or somewhat against the Indian Army.

Now, a provenance graph of this suspicious tweet as highlighted in Figure 4.21 is required to visualize and analyze to obtain *Who*, *Why*, and *How* Provenance such as *Who* has posted this tweet?, *Why* this is present in the result set?, and *How* it is derived? etc. A partial provenance graph of above example query 5 is shown in Figure 4.22, where provenance information of all the result tuples i.e. grey nodes, are shown with their respective provenance edges, those are associated with a set of nodes of the source graph

"TWEET"
Boys played well.. #IndianArmy @adgpi should roast/kill #Kashmiri stone pelters who protect #Militants and #Terrorists
Leaders of different groups in #Kashmir are responsible for the death of youth there. They never ask #Kashmiris to keep away from sites where #IndianArmy fights #Terrorists. Those who reach encounter sites are likely to be killed or injured ie. collateral damage
Nearly two dozen civilians hit by bullets and rest hit by pellets. Another bloody day in IAK.
Barbarian #IndianArmy kills more civilians in #Kashmir today, it,s sorry how world is silent about #Kashmir conflict @UN and frustrated #India kills innocent civilians @EUatUN @UKParliament @EUCouncil @IndiaUNNewYork @amnesty @OICatUN @IntlCrimCourt #Pulwamakillings #Genocide https://t.co/ZSS1AWA5DL
"#IndianArmy have killed 10 Kashmiri civilians and over 5 dozen Civilians also injured, during an operation in #Pulwama District, Jammu & Kashmir. #Kashmir #KashmirBleeds"
@Jitin_Tweets @AltafQu85217484 News flash wen floods came in #Kashmir #IndianArmy were airlifting VVIP and tourists, they left the civilians stranded and #Kashmiris helped their own. This is a fact!
#IndianArmy in its fresh act of state terrorism killed a number of civilians. #India refuse to resolve #kashmir dispute, @PMOIndia try to crush peoples, desire of #Right to self-determination by killing civilian @EUatUN @EUCouncil @Turkey_UN @UKParliament https://t.co/QyxjC8Nhh7

Figure 4.21: Snapshot of Result of Example Query 5

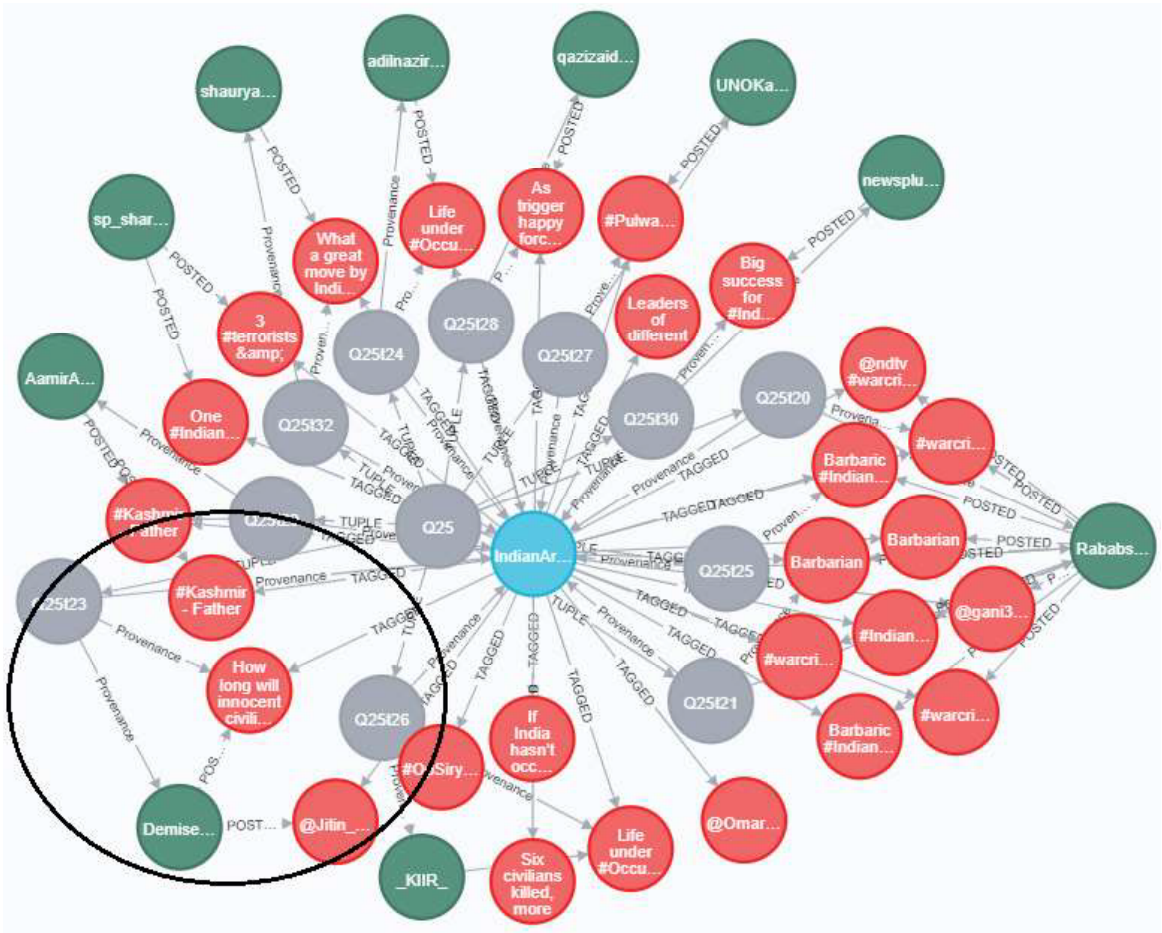


Figure 4.22: Partial Provenance Graph of Example Query 5

which contributed to produce it. The provenance graph of this identified suspicious tweet with result tuple id Q25t26 is mentioned with a marked portion in Figure 4.22. After visualizing this provenance graph, we determined that this suspicious tweet is posted by a twitter user whose screen name is 'Demise_ _ _ _' as shown in Figure 4.23. Now, on the basis of this screen name, we retrieved all the other tweets posted by this user in the source graph database as shown in Figure 4.24.

Again, analyzing all the other tweets posted by this identified user during Pulwama incident held on 15/12/2018, we found that tweet's texts are looking suspicious or somewhat against the Indian Army and our country, which may provoke others also. Therefore, there is a need to retrieve all the tweets posted by this suspicious user on twitter's timeline along with his attributes such as Profile Creation Date, Nationality, Location, Sex, Friend's List, Follower's List etc., and his linked communities on Twitter and other social networking sites, for further analysis.

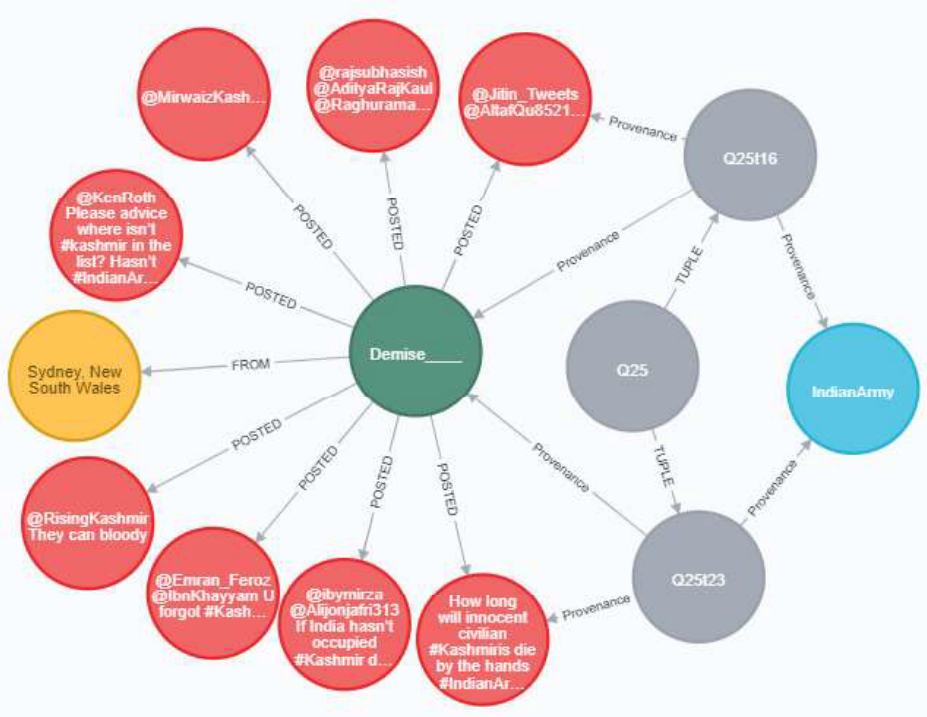


Figure 4.23: Partial Provenance Graph

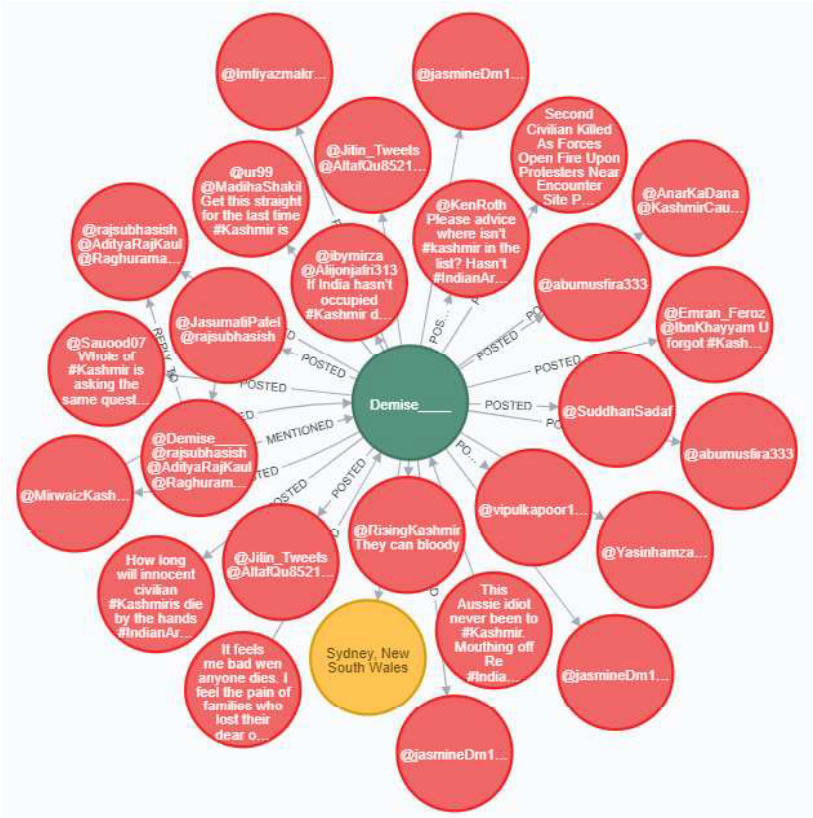


Figure 4.24: All Tweets Posted by Identified User 'Demise_ _ _ _'

Ultimately, the proposed framework works in following four phases for the above cyclic process model; First, to generate provenance graphs for all the result tuples. Second, determine suspicious or provoking tweets exist in the result set. Third, identify suspicious person who posted those tweets. Fourth, retrieve all the attributes of that suspicious person and his/her linked communities. In this way, framework has the capability to model a large twitter dataset into a Neo4j graph database to perform provenance visualization based on suspicious tweets and hashtags, which can be helpful for security agencies in investigating suspicious persons and their linked communities. Also, it will be helpful in attempting to understand the attitude and behaviour of social media users. The framework supports following key features viz. big data modelling, social data analytics, exploration and visualization etc. It can also be used to extract intelligence from social networking sites.

4.7 Conclusions and Future Work

In this chapter, we designed and implemented a Zero-Information Loss Graph Database (ZILGDB) on top of which a Social Data Provenance Framework is developed to capture provenance information for Twitter data set. The proposed framework has the capability to capture provenance information for a query set including select queries, aggregate queries, and range queries with timeline. It also supports historical data queries as well as capturing provenance information for historical queries, using updates management in ZILGDB. However, a small overhead is associated with provenance capturing of aggregate queries, and queries having larger number of result tuples as compared to queries having lesser number of tuples in their result set. Proposed framework provides support for efficient provenance querying for both justifying answers of a query result, and historical data queries. The framework also supports multi-depth querying of provenance data in graph database to know about the direct/indirect sources of any information. In this way, generating social provenance in a detailed visual form can help in analyzing a huge social network for critical decision making and in strategic planning. Our proposed framework and provenance algorithms prove to be very promising; offering an effective approach to address increasingly challenging issue of trust in social media. We conducted a real life

use case study to evaluate the usefulness of such detailed social provenance generated by our framework in terrorist attack investigation, to identify suspicious persons and their linked communities on social media platform, particularly in Twitter's network. In the future, we also plan to extend our framework for distributed graph database on different nodes in a cluster.